Jonathan Shao-Kai Huang

December 9, 2025

Abstract

This is a LaTeXnote that started during the 2025 ASIoP summer internship.
2025 ASIoP Summer Program
SATORI Group

Shao-Kai Jonathan Huang[1]

[1]) Department of Physics, National Taiwan University; SATORI (Synergy of Algorithms, Theory and Observations to Reveal the Invisible), Academia Sinica

Contents

Chapter 1

Introduction to Machine Learning

Machine Learning (ML) is a study subfield of **Artificial Intelligence (AI)** focusing on the development and research of statistical algorithms that can learn data, with the ultimate goal of performing certain tasks without explicitly programmed instructions. Statistics and mathematical optimization comprise the foundations of machine learning. The term machine learning was termed by AI and computer gaming pioneer, Arthur Samuel in 1959. In the 1950s, Samuel made a computer checker program that calculated the winning chance of both sides, the first ML algorithm to appear.

> **Note** (Machine Learning in the Eye of a Mathematician). Tom M. Mitchell provided a more formal definition of ML algorithms: *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."* This is in agreement with Alan Turing's paper "*Computing Machinery and Intelligence*", where the question *"Can machines think?"* is replaced with the question *"Can machines do what we (as thinking entities) can do?"*

The computational analysis of machine learning algorithms and their performance is known as computational learning theory via the **probably approximately correct learning model**.

1.1 Models of Machine Learning

A **machine learning model** is a type of mathematical model that can be used to make predictions or classifications on new data once trained on a given training dataset. There are many types of ML models, which we will list below.

1.1.1 Support-Vector Machines

Support-vector machines (SVMs) are a set of supervised learning algorithms used for classification and regression. An SVM algorithm is a non-probabilistic, binary, linear classifier, though nonlinear classification can be performed with the **kernel trick**. Given a training dataset with two classification labels, a SVM algorithm builds a model predicting whcih category a new example falls into.

1.1.2 Regression Analysis

1.1.3 Decision Trees

Decision tree learning uses a decision tree as a predictive model to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). When decision trees admit discrete or continuous values for target values, they are called **classification trees** or **regression trees**, respectively. A learning model based on decision trees is random forest regression (RFR), which builds multiple (independent) decision trees and takes the average of their predictions, thus improving accuracy.

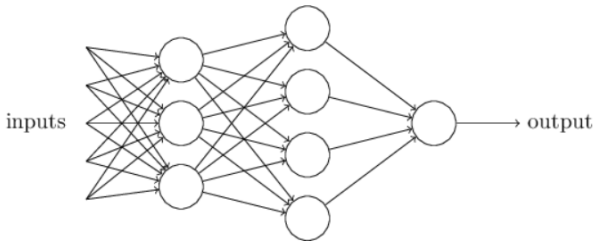

Figure 1.1: Enter Caption

1.1.4 Genetic Algorithms

A **genetic algorithm (GA)** is a search algorithm that mimics the process of natural selection, using methods such as mutation and crossover to generate new genotypes. The goal of GAs is to find good solutions to a given problem, and they have been used in the field of ML since the 1980s.

1.1.5 Neural Networks

Advances in the field of deep learning, a subfield of machine learning based on neural networks, has allowed these neural networks to surpass many machine learning approaches. Next we talk more about neural networks and their working principle.

"In machine learning, a *neural network* (also *artificial neural network*) is a computational model inspired by the structure and functions of biological neural networks." (Wikipedia - Neural network (machnie learning)) These networks are typically trained through empirical risk minimization, and are the basic model for machine learning.

An easy question when one hears about the breath-taking architecture of neural networks is: why neural networks? Well, the study of artificial neural networks began as an attempt to adopt the complex structures of the human brain to perform tasks that conventional algorithms had trouble with. Since the *neurons* are connected in diverse ways, NNs have the remarkable ability to **learn** and model *non-linearities* in complex systems.

A NN consists of *simulated neurons*, which admits a value in the interval $[0, 1]$. Each neuron is connected to other nodes via edges, just like the structure of a biological **axon-synapse-dendrite** connection, forming a directed, weighted graph.

1.1.6 Perceptron and Multi-Layer Perceptron

The main reference for this section is the well-written online book http://neuralnetworksanddeeplearning.com/chap1.html. In much modern works, the neuron used is a *sigmoid neuron*. However, we shall begin with one of the earliest model of an artificial neuron, the *perceptron*, as an introduction. The perceptron was developed in the 1950s by Frank Rosenblatt; a perceptron takes several binary inputs, $x_1, x_2, \ldots$, and produces a single binary output. He introduced weights, $w_1, w_2, \ldots$, real numbers expressing the importance of the respective inputs to the output. The perceptron's output was restricted to 0 or 1 only, and is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some threshold value. This is more similar to the **all-or-none principle** in physiology. It should seem plausible that a complex network of perceptrons could make quite subtle decisions:

We can introduce this with an example inspired by 3B1B's online lecture on deep learning.

Since it is cumbersome to write

$$a_0^{(1)} = \sigma\left(w_{0,0}^{(0)} a_0^{(0)} + w_{0,1}^{(0)} a_1^{(0)} + \cdots + w_{0,n}^{(0)} a_n^{(0)} + b_0^{(0)}\right), \tag{1.1}$$

for every neuron, we can collect the neuron values into a vector, and organize the weights correspondingly into a matrix (second-rank tensor). Then the right hand side becomes

$$\begin{pmatrix} w_{0,0}^{(0)} & w_{0,1}^{(0)} & \cdots & w_{0,n}^{(0)} \\ w_{1,0}^{(0)} & w_{1,1}^{(0)} & \cdots & w_{1,n}^{(0)} \\ \vdots & \ddots & \vdots & \vdots \\ w_{k,0}^{(0)} & w_{k,1}^{(0)} & \cdots & w_{k,n}^{(0)} \end{pmatrix} \begin{pmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix}. \tag{1.2}$$

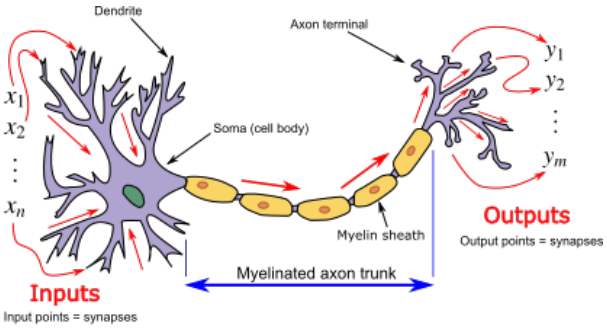

Figure 1.2: Diagram of a typical neuron in biology.

More compactly, let

$$\mathbf{a}^{(0)} = \begin{pmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix}, \ \mathbf{a}^{(1)} = \begin{pmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_k^{(1)} \end{pmatrix}, \ \mathbf{W} = \begin{pmatrix} w_{0,0}^{(0)} & w_{0,1}^{(0)} & \cdots & w_{0,n}^{(0)} \\ w_{1,0}^{(0)} & w_{1,1}^{(0)} & \cdots & w_{1,n}^{(0)} \\ \vdots & \ddots & \vdots & \vdots \\ w_{k,0}^{(0)} & w_{k,1}^{(0)} & \cdots & w_{k,n}^{(0)} \end{pmatrix}, \tag{1.3}$$

so

$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}\mathbf{a}^{(0)} + \mathbf{b}^{(0)}\right). \tag{1.4}$$

Therefore, instead of thinking about neuron as objects that hold a number, we should see them as a function of the outputs of the previous layer which outputs another value between 0 and 1.

1.1.7 Activation Functions

Activation functions introduce nonlinearities into the theory, which is vital for learning to take place. Famous activation functions include ReLu and the sigmoid function. As mentioned above, we want a function that "emphasizes" the values of neurons near the features we are interested in.

For example, to recognize shape in the top-right corner, we may assign large weights to neurons there, negative weights along its border, and near-zero weights everywhere else. To enlarge this effect, we put the resulting vector $\mathbf{a}^{(0)}$ along with some bias vector $\mathbf{b}$ into a specially designed function, called an **activation function**. The **sigmoid** or **logistic** function, first introduced by Hinton et al. (2012) for speech recognition, tends to 0 when input is negative, tends to 1 when input is positive, and increases steadily near zero input value; the **ReLU** function, first introduced in 2012 in the AlexNet computer vision model, is zero whenever the input is negative, and increase linearly for positive input.

Looking at the dates above, we see that the study of neural networks is a new and growing field! Other than the ones mentioned above, other activation functions include **GELU** (2018), **SiLU**, **ELU**, **LeakyReLU**, **Softplus**, and **tanh**. These activation function have explicit derivative forms.

Finally, **Softmax** is an activation function that takes input from more thah one preceeding layers. It is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

1.1.8 Backpropagation

These include stochastic gradient descent (SDG), Adam, etc,

1.2 What is Learning?

Learning is the adaptation of the network to better handle a task by considering sample observations. In the context of neural networks, learning involves adjusting the weights (and optional thresholds) of the network to improve the accuracy of the result. This is done by minimizing the observed errors, or, the loss function (see previous subsection **??**).

1.2.1 Learning Paradigms

As discussed above, a neural network has a large number of parameters, including weights and biases, that have to be tweaked in order to make the best prediction. **Learning** is *the process of finding the right weights and biases*. There are three main paradigms of learning: **supervised learning**, **unsupervised learning**, and **reinforcement learning**.

Supervised Learning

Unsupervised Learning

Tasks that fall within the paradigm of unsupervised learning are in general *estimation problems*. Applications include clustering, the estimation of statistical distributions, compression, and filtering.

Reinforcement Learning

1.2.2 Training and Cross-Validation

Train/test split. We simulate a collection of data, and split it into a training set and a test set. Usually, the test set is smaller, but they are both uniformly distributed.

1.2.3 Overfitting and Underfitting

Overfitting means that the model is memorizing the training data rather than learning generalizable patterns. This means that the model has learned the training data *too well*, effectively memorizing the training data set, and thus struggles to **generalize**.

1.3 Loss

In machine learning, **training loss** and **validation loss** are two critical metrics used to evaluate a model's performance during the training process. In the project, various training and validation loss trends are plotted using `tensorboard`.

- Training loss: Training loss measures the error of the model on the training data. It indicates how well the model is fitting the data it has seen, and its decrease generally suggests effective learning of patterns within the training set. During training, the goal is to minimize this loss, however, a very low training loss can indicate overfitting.
- Validation loss: Validation loss measures the error of the model on the validation data, which is a separate dataset that the model has not seen during training. If the training loss continues to decrease but the validation loss starts to increase, it is a strong indicator of overfitting.

There are a few commonly used loss functions, which we will discuss in detail below.

1.3.1 Mean Square Error

1.3.2 Cross Entropy Loss

Cross-entropy loss is a commonly used loss function in classification problems, especially in logistic regression and neural networks. It measures the difference between the true probability distribution and the distribution predicted by the model, quantifying how well the output matches the true labels.

The cross entropy loss is non-negative, and has high sensitivity to confidence, i.e. it penalizes confident but incorrect predictions more heavily than unconfident ones.

1.3.3 Gaussian Loss

1.3.4 Heteroscedastic Gaussian Loss

Heteroscedastic Gaussian Process Regression is an algorithm to estimate simultaneously both mean and variance of a non parametric regression problem. Unlike standard *Gaussian Process regression* or *SVMs*, it is able to estimate variance locally. [LSC05]

This is the loss function currently used in the DNN training process for `energy` and `xmax`.

1.4 Latency

1.5 Bias-Variance Tradeoff

1.5.1 James-Stein Estimator

A famous demonstration of the bias-variance tradeoff is the **James-Stein Estimator**.

The idea of shrinkage is known as **regularization** in the context of machine learning. As the number of parameter grows, adding a shrinkage factor can sometimes reduce the mean square error of our estimation. Shrinking different parameters separately can also give us an idea of which parameters are important.

1.6 State of the Art of Machine Learning

After the success of **AxiNet**, the focus of machine learning research shifted towards deep learning, which will be discussed in detail in the next section.

Chapter 2

Deep Learning

2.1 What is Deep Learning

This section will use [20] and () as the main reference.

Deep learning (DL) is the study of *hierarchically-structured* function approximators known as *neural networks*, which was introduced in subsection 1.1.5.

2.1.1 Historical Development

The first working deep learning algorithm was the Group method of data handling, a method to train arbitrarily deep neural networks (Ivakhnenko & Lapa, 1965). The first deep learning multilayer perceptron trained by stochastic gradient descent (SDG) was published in 1967 by Shun'ichi Amari.

In a deep neural network, the circuits, or the neural networks, usually have many layers. In 1943, Warren McCulloch and Walter Pitts formulated the **McCulloch-Pitts (MCP) neuron** in their paper "*A logical calculus of the ideas immanent in nervous activity*" [MP43], shown in figure (2.1). The MCP neuron is a restricted artificial neuron introduced during the formative years of ANNs (1943 - 1958) as a way to model biological networks in the brain. Nonlinearities were modeled by introducing a threshold logic function, with inhibitory and excitatory inputs and bias.

In 1958, **Franklin Rosenblatt** introduced a major advancement called the **perceptron**, now regarded as the first generation neural networks [Man20]. Based on the MCPs neuron and the findings of Canadian psychologist **Donal O. Hebb**, Rosenblatt developed the first perceptron, with learnable weight and bias values that could be determined analytically or by a *learning algorithm*. Rosenblatt's perceptrons paved the way for the development of ANNs, which progressed into deep learning with its applications in AI.

The Rosenblatt Perceptron consisted of a single-layer MCP model neuron which sums up all the weighted inputs and produces a binary output using a threshold activation function. Such networks and their variations are collectively called perceptrons. Firthermore, Rosenblatt proved that if the input vectors representing two linearly separable classes are used to train the model neuron, the perceptron learning algorithm will always converge.

The real reason for the success of deep learning has not yet been elucidated. It is plausible that the long computation paths present in a DNN allow input variables to interact in complex ways.

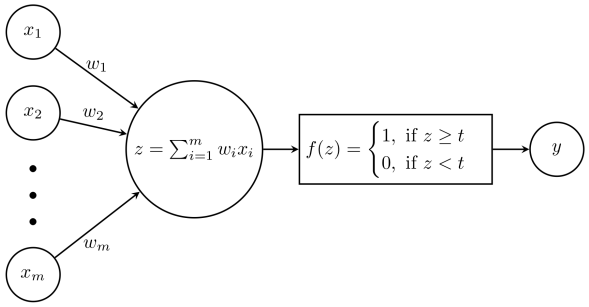

Figure 2.1: Diagram of a McCulloch-Pitts neuron

Definition 2.1.1 (Units). Each node in a neural network is also called a *unit*.

Definition 2.1.2 (Deep Learning). We can define deep learning (DL) as follows:

- We call a model a deep model if back-propagation is required to train it.
- A network is composed of layers of nodes, normally a neural network is called a deep neural network (DNN) if it has many layers.
- Deep learning is a subset of machine learning that is based on deep models, and learns using a deep neural network.

Theorem 2.1.1 (Universal Approximation Theorem). A network with two or more layers and at least one nonlinear unit can be utilized to approximate any continuous function.

Remark. A deep learning model is a function approximator.

The above remark can be illustrated with the folowing example.

Example (A neural network corresponds to a function mapping). The value of the j-th unit in the i-th layer, where the input layer is labelled with $i = 0$, is represented with $x_j^{(i)}$. Then in the first layer, we have

$$\begin{aligned} x_1^{(1)} &= g_1^{(1)}\left(w_{11}^{(1)}x_1^{(0)} + w_{21}^{(1)}x_2^{(1)} + b_1^{(1)}\right) = g_1^{(1)}\left(\mathbf{w}_1^{(1)} \cdot \mathbf{x}^{(0)} + b_1^{(1)}\right), \\ x_2^{(1)} &= g_2^{(1)}\left(w_{12}^{(1)}x_1^{(0)} + w_{22}^{(1)}x_2^{(1)} + b_2^{(1)}\right) = g_2^{(1)}\left(\mathbf{w}_2^{(1)} \cdot \mathbf{x}^{(0)} + b_2^{(1)}\right). \end{aligned} \tag{2.1}$$

This can be collected into a clean vector equation:

$$\mathbf{x}^{(1)} = \mathbf{g}^{(1)}\left(\mathbf{w}^{(1)} \cdot \mathbf{x}^{(0)} + \mathbf{b}^{(1)}\right). \tag{2.2}$$

For sake of clarity, we will use normal math font instead of bold font to represent these vectors. This gives

$$\begin{aligned} x^{(1)} &= g^{(1)}\left(w^{(1)} \cdot x^{(0)} + b^{(1)}\right), \\ x^{(2)} &= g^{(2)}\left(w^{(2)} \cdot x^{(1)} + b^{(2)}\right) \\ &= g^{(2)}\left(w^{(2)} \cdot g^{(1)}\left(w^{(1)} \cdot x^{(0)} + b^{(1)}\right) + b^{(2)}\right). \end{aligned} \tag{2.3}$$

Writing the input layer $x^{(0)}$ simply as x, and this network corresponds to the map

$$h_W(x) = g^{(2)}\left(w^{(2)} \cdot g^{(1)}\left(w^{(1)} \cdot x + b^{(1)}\right) + b^{(2)}\right). \tag{2.4}$$

We generalize the above discussion to more complex DNNs: the collection of weights can be organized as a tensor

$$W = \begin{pmatrix} w^{(1)} & w^{(2)} & \cdots & w^{(m)} \end{pmatrix}^T \in M_{m\times n}(\mathbb{R}), \tag{2.5}$$

the bias is a vector $b^{(i)} \in \mathbb{R}^m$, and the recursive relationship between neurons $x^{(i-1)} \in [0,1]^n$ and $x^{(i)} \in [0,1]^m$ of neighboring layers is

$$x^{(i)} = g^{(i)}\left(w^{(i)} \cdot x^{(i-1)} + b^{(i)}\right). \tag{2.6}$$

2.1.2 End-to-End Learning

Many times, a complex computational syste, for some task is composed of several trainable subsystems, so it is helpful to train these subsystems and connect their output to the input of the next subsystem, reducing training capacity requirements. There are important distinctions between each of the three important components of a DNN: **input**, **output**, and **hidden layers**.

1. Input layer: This layer encodes training or testing data into numerical values to be associated with input units.
2. Output layer: Ideally, the loss for output units would be zero or close to zero. However, this is almost never the case for randomly chosen parameters, so we have to adjust these parameters using backpropagation and have the quality of parameter estimation after each application evaluated with some loss function. A common loss function is the **negative log likelihood**.
3. Hidden layer: During the years 1985 to 2010, the sigmoid and tanh functions were exclusively used for the hidden layers, while from 2010 onwards ReLU and softplus are used. It is observed (but not yet rigorously explained) that *deep and narrow networks* usually have better learning ability compared with shallow and wide networks when the total number of weights is kept fixed.

Definition 2.1.3 (Negative Log Likelihood and Entropy)**.** Let x_j be the N examples given as input to the DNN, and y_j the respective true value. We shall minimize the function

$$-\sum_{j}^{N} \log\left[P_W\left(y_j|x_j\right)\right]. \tag{2.7}$$

This is an approximation to the **cross entropy loss**, defined over two probability distributions P, Q and given by

$$H(P,Q) = \mathbb{E}_{z\sim P(z)}\left[\log Q(z)\right] = \int \mathrm{d}z\, P(z) \log Q(z). \tag{2.8}$$

Minimizing the cross entropy loss of our probability P_W would be equivalent to minimizing

$$H\left(P^*(x,y), P_W\left(y|x\right)\right), \tag{2.9}$$

but it is impossible to know the true distribution *a priori* (otherwise there won't be so much trouble!), so we go with the previously mentioned approximation.

Remark (KL Divergence)**.** The cross entropy of two probability distributions is not a distance function (metric) in the space of probability distributions. This can be seen by the fact that

$$H(P,P) = \int \mathrm{d}z\, P(z) \log P(z) \equiv H(P) \neq 0. \tag{2.10}$$

Here $H(P)$ is the familiar Shannon entropy. However, we *can* define a metric by subtracting off the nonzero part, which gives the definition of the **Kullback-Leibler divergence**:

$$D_{\mathrm{KL}}\left(P\,||\,Q\right) \equiv H(P,Q) - H(P). \tag{2.11}$$

2.2 Recent Trends

In the 2025 COLT (Conference On Learning Theory, one of the most prominent pure theory conferences in the field of ML), the top five most popular research categories are, neural networks/deep learning, online learning, bandit problems, reinforcement learning, and high-dimensional statistics. This is taken from Li Yen-Huan's Facebook post from 2025 July.

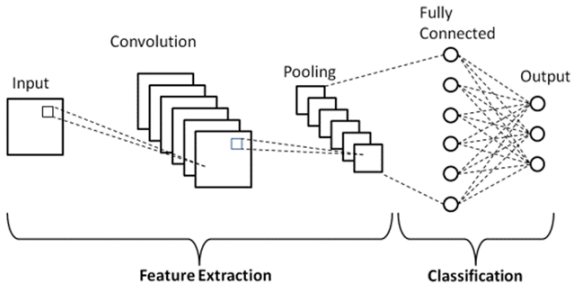

Figure 2.2: The kernel is a fixed pattern of weights that convolutes with every region of the input image.

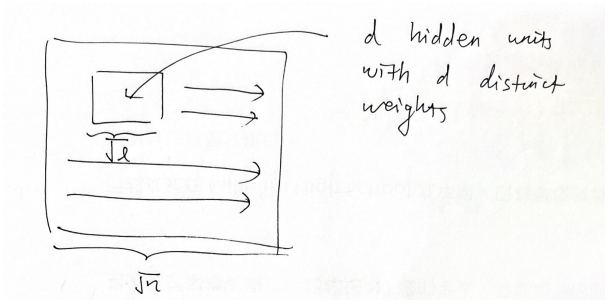

Figure 2.3: The kernel is a fixed pattern of weights that convolutes with every region of the input image.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special type of deep neural networks (DNN) consisting of three main steps: **convolutional kernel**, **pooling**, and **vetorization**. CNNs are also one of the first ANNs that closely mimic the biological working of *visual neurons*, and have become an essential tool for computer vision.

In order to classify images, we want to follow the following rules of thumb:

Note (Rule of Thumb 1)**.** Construct the first hidden layer such that each hidden layer unit receives input from a small local region. That is, let each local region possess $l \ll n$ pixels, then the total would be $N = ln \ll n^2$.

Note (Rule of Thumb 2)**.** We expect image data to exhibit approximate spatial invariance, meaning that the image of an object translated by some distance is the same distance.

Remark. The above two rules of thumb do not capture the *rotational invariance* of an object, hence any rotation of an image of a dog may render it completely unrecognizable to the algorithm. To solve this problem, we can naively give the CNN a training set containing rotated images.

By injecting some prior knowledge of adjacency and spatial invariance, we can develop models with far fewer parameters that can recognize images.

2.3.1 Introducing the Kernel

The **kernel** is a small local region that sweeps over the image. This ensures translational invariance is obeyed by duplicating the same node structure over the entire image. Refer to figure (2.3), suppose as before that the 2D image contains n pixels, the kernel contains l pixels and has d hidden units with distinct weights. In total there are dl weights, a number independent of the image size! We can understand the kernel as the *fixed pattern of weights* that scans through the input images.

Example (1D Convolutional Neural Network)**.** The idea of a CNN kernel is best illustrated with a 1D

problem. Supppose there are $n = 8$ units in the input layer, and the kernel consists of $l = 3$ units, with stride $s = 1$. Then the network produces

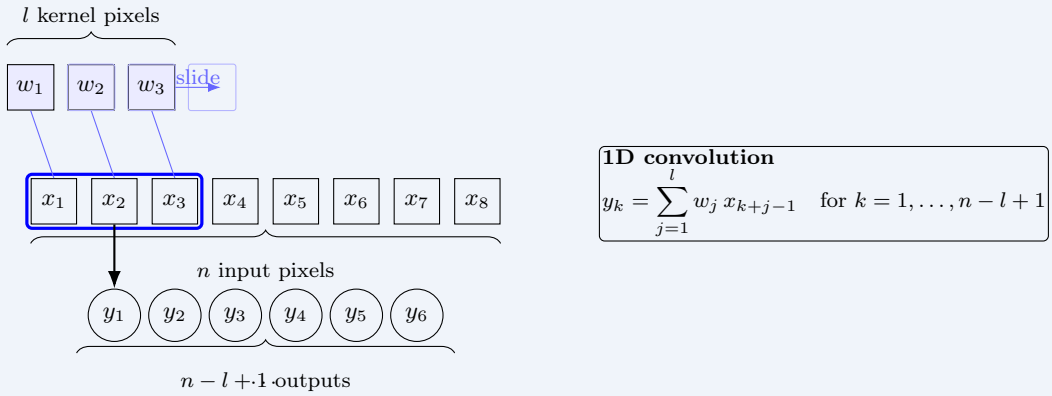

Figure 2.4: Illustrating the working principles of a DNN kernel with a one-dimensional CNN.

Example (An Example with Numbers). Consider the 1D CNN problem with kernel size $l = 3$ and stride $s = 2$ in the following figure. Suppose the values in the nodes represent how bright the pixel is, then the specific kernel we have chosen picks out region with low intensity, i.e. the dark spots in the image. As the result indicates, the central region with a 2 in the middle returns the highest value (9) after convolution with the kernel.

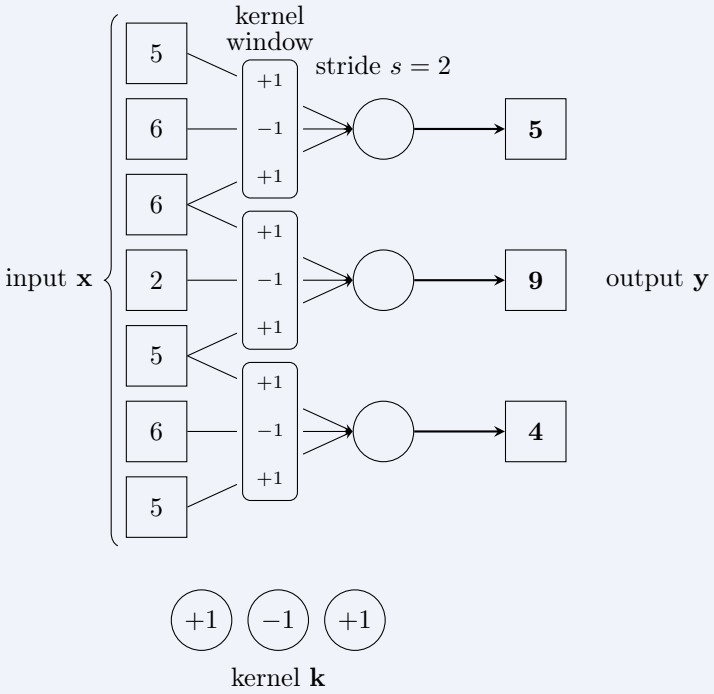

Figure 2.5: Numeric example of 1D convolutional network.

The **discrete convolution** can be written as

$$z = x * k, \tag{2.12}$$

which in fact means

$$z_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2}. \tag{2.13}$$

Although not necessary, the above convolution process can be represented as matrix multiplication, with the kernel matrix K constructed by translating the kernel k and filling the remaining entries with zeroes:

$$\begin{pmatrix} 1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \\ 6 \\ 2 \\ 5 \\ 6 \\ 5 \end{pmatrix} = \begin{pmatrix} 5 \\ 9 \\ 4 \end{pmatrix}. \tag{2.14}$$

Here the output is $\text{out}(x) = (5, 9, 4)$. In higher-dimensional CNNs this equation is appropriately generalized into **tensor multiplication** [a].

[a] In ML lingo, a tensor just represents any tuple or high-dimensional array of numbers, and is distinct from tensor as defined in mathematics, which are basis-independent algebraic objects that describes a multilinear relationship between sets of objects living inside a vector space.

Note (Computation for CNNs: GPU vs. TPU). **Tensor Processing Unit (TPU)** is an application-specific integrated circuit (ASIC) developed by Google for NN learning, using Google's own TensorFlow software. When using stochastic gradient descent (SGD), we separate data into **minibatches** chosen randomly ("stochastically") at each step. Each training example in the SGD minibatch can be computed in parallel, so we can leverage the power of hardware parallelism with TPUs.

2.3.2 CNN is Like a Visual Cortex

CNNs were originally inspired by visual cortex models developed in neuroscience. A node in the m-th hidden layer has a **receptive field** of size $(l-1)m+1$, and with stride s the receptive field grows as $O(ls^m)$, indicating exponential growth with respect to depth.

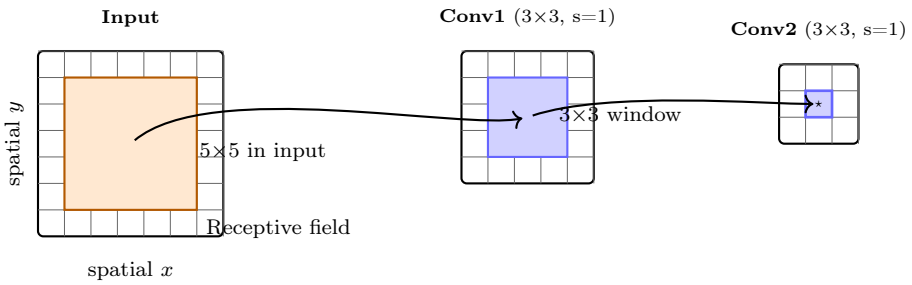

Figure 2.6: The receptive field in a 2D convolutional network is shown.

2.3.3 Pooling and Downsampling

To achieve *translation invariance* and reduce spatial dimensionality, CNNs interleave convolutions with *pooling* (e.g. max or average pooling). The pooling layer summarizes a set of adjacent nodes from the previous (convolutional) layer with a single value, and there are two common ways to get this value: **average-pooling** and **max-pooling**.

- Average-pooling takes the kernel $k = (1/l, \ldots, 1/l)$ with size l and convolutes with the previous layer, returning the average of all the nodes. This is useful for multiscale recognition.
- Max-pooling returns the maximal value from the l input nodes, discarding everything else.

Pooling is a fixed layer and acts like a convolution layer as described above, reducing layer size, i.e. **downsmapling**.

Example (Tensor Operations). In ML lingo, a tensor operation is the manipulation of high-dimensional arrays of numbers. If we use a 256×256-pixel RGB image as input example, with minibatch size 64,

then out input is a (fourth-rank) tensor of the shape $256 \times 256 \times 3 \times 64$. After convolution with 96 kernels, each with size $5 \times 5 \times 3$ and stride 2, the resulting tensor has the shape $128 \times 128 \times 96 \times 64$.

We call these 96 kernels **channels**, and they form a **feature map** representing different information about some specific feature. In a very deep neural network (such as this one), later kernels will generally represent high-level features, such as contour or faces in a facial recognition program.

2.3.4 Batch Renormalization

It is observed that by rescaling internal values of the CNN, we can improve the rate of convergence of SGD. The reason for this is not well-understood.

For learnt parameters β and γ, which may be different for different nodes (so they are *node-specific*), we will renormalize each output from the i-th node as

$$z_i \longrightarrow \hat{z}_i = \gamma \frac{z_i - \mu}{\sqrt{\epsilon + \sigma^2}} + \beta, \tag{2.15}$$

where $\mu = \text{mean}(z)$, $\sigma = \text{STD}(z)$, and $0 < \epsilon < 1$ is a small parameter to avoid division by zero.

2.3.5 Regularization

Regularization is a method used to limit the complexity of a model.

In the context of ANNs, regularization is normally implemented with **weight decay**. With weight decay, we penalize large weights, so the CNN must be more confident when choosing large weights. This prevents arbitrarily odd behavior that come from overfitting. To implement weight decay, we add the **penalty function**

$$P(W) = \lambda \sum_{i,j} |W_{ij}|^2 = \lambda \|W\|^2 \tag{2.16}$$

to the loss function, i.e.

$$\text{Loss} \mapsto \text{Loss} + P(W). \tag{2.17}$$

Here λ is a *hyperparameter* usually chosen to be around 10^{-4}.

Note. When we choose a network architecture, the choice is usually arbitrary. For example, the number of layers and the number of nodes within each individual layer are usually not chosen based on some first principles. This imposes an absolute constraint on our hypothesis space. However, weight decay acts as a penalty, and therefore gives a softer constraint on the network.

2.3.6 Dropout

Dropout is a *regularization technique* to increase the generalization power of the DNN model. During training, we deactivate a randomly chosen fraction p of nodes and scale the surviving nodes accordingly, i.e. each node z_i is either set to z_i/p with probability p or 0 with probability $1-p$. Then, backpropagation is applied to the newly created neural network of expected size pn, where n is the original size of the DNN. Refer to the diagram below for an illustration.

Empirically it has been observed that $p = 0.5$ works best for hidden layers, while $p = 0.8$ works best for the input layer. Since dropout introduces noise during training, it makes the model more robust, hence better at generalizing to more data. The downside is of this is higher difficulty in training.

Remark. (i) The dropout scheme is similar to selection processes that guide gene evolution in nature.

(ii) Applying dropout to later layers in the CNN forces the output to be produced robustly, since the network is trained to pay attention to all the high-level features in the input examples.

Chapter 3

Large Language Models

Large Language Models (LLMs) are the core of modern day artificial intelligence and intelligent models such as ChatGPT (OpenAI), Claude (Anthropic), Gemini (Google), Grok (xAI), Llama (Meta), and TAIDE (NSTC). This section, we will briefly discuss the working principles of LLMs and the reason for their success.

3.1 Attention Mechanism

3.2 Generative Pretrained Transformers

3.3 Small Language Models

In contrast with the popular LLMs, small language models (SSMs) that require less parameters have also been intensively developed.

Chapter 4

Graph Neural Networks

4.1 Graphs

4.2 Graph Neural Networks

4.3 Graph Convolutional Networks

4.4 Message-Passing Neural Networts

4.4.1 Graph Attention Networks

Graoh Attention Networks (GANs) work by sending the ? into an attention function.

4.5 Summary

Graph neural networks are very general and powerful frameworks for machine learning. For example, not only is convolutional neural networks a special case of a GNN, an LLM is also a special case of a GNN. This fact is seen by

Chapter 5

Coding in the Age of AI

5.1 Vibe Coding

5.1.1 What is vibe coding?

"Vibe coding" is introduced by renowned Computer scientist Andrej Karpathy in February 2025 and emphasized the significance of AI tools in software development.

5.2 Context Engineering

Context engineering is building dynamic systems to provide the right information and tools in the right format such that the LLM can plausibly accomplish the task. Most of the time when an agent is not performing reliably the underlying cause is that the appropriate context, instructions and tools have not been communicated to the model. LLM applications are evolving from single prompts to more complex, dynamic agentic systems. As such, context engineering is becoming the most important skill an AI engineer can develop.

Definition: Context engineering is building dynamic systems to provide the right information and tools in the right format such that the LLM can plausibly accomplish the task.

5.2.1 ChatGPT and Where It Is Going

The length of tasks achievable by AI agents with at least 50% success rate has been exponentially increasing, as shown in figure 5.3. This is a result of improvements in AI models and the development of more sophisticated tools that can be integrated into AI workflows.

5.3 Context Engineering is not Prompt Engineering

Why the shift from "prompts" to "context"? Early on, developers focused on phrasing prompts cleverly to coax better answers. But as applications grow more complex, it's becoming clear that providing complete and structured context to the AI is far more important than any magic wording.

5.4 Some Warnings

Research has shown that for experienced programmers, AI assistance actually *slows down* productivity.

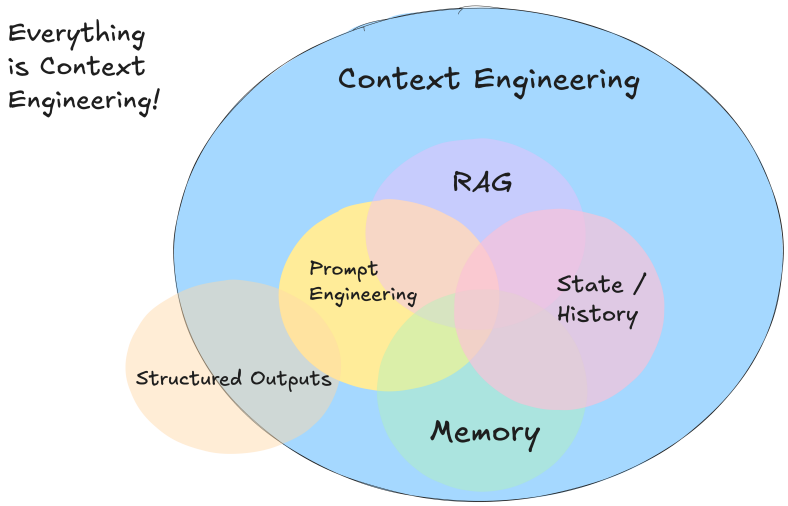

Figure 5.1: Context engineering diagram from https://blog.langchain.com/the-rise-of-context-engineering/

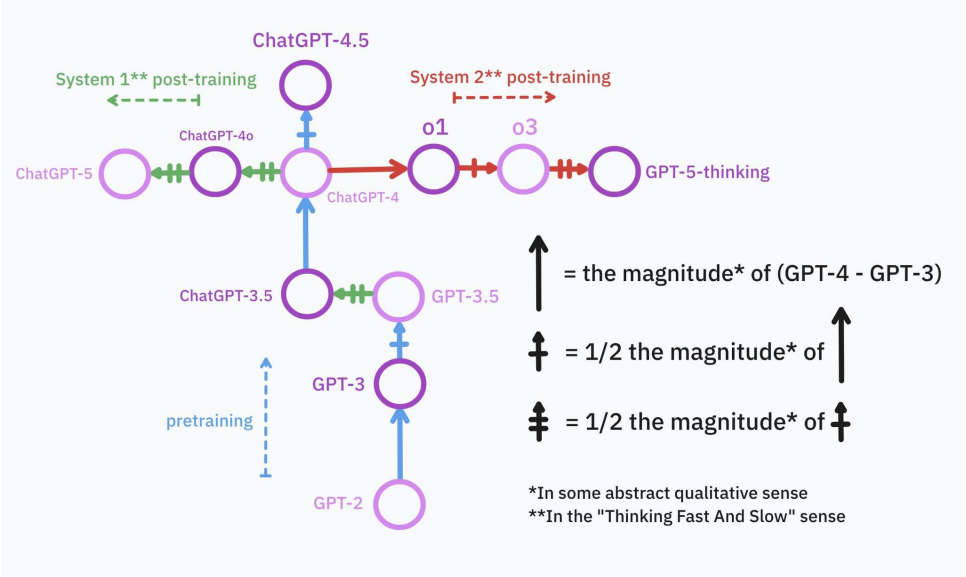

Figure 5.2: A schematic diagram of the relation between various ChatGPT models by X user @arithmoquine.

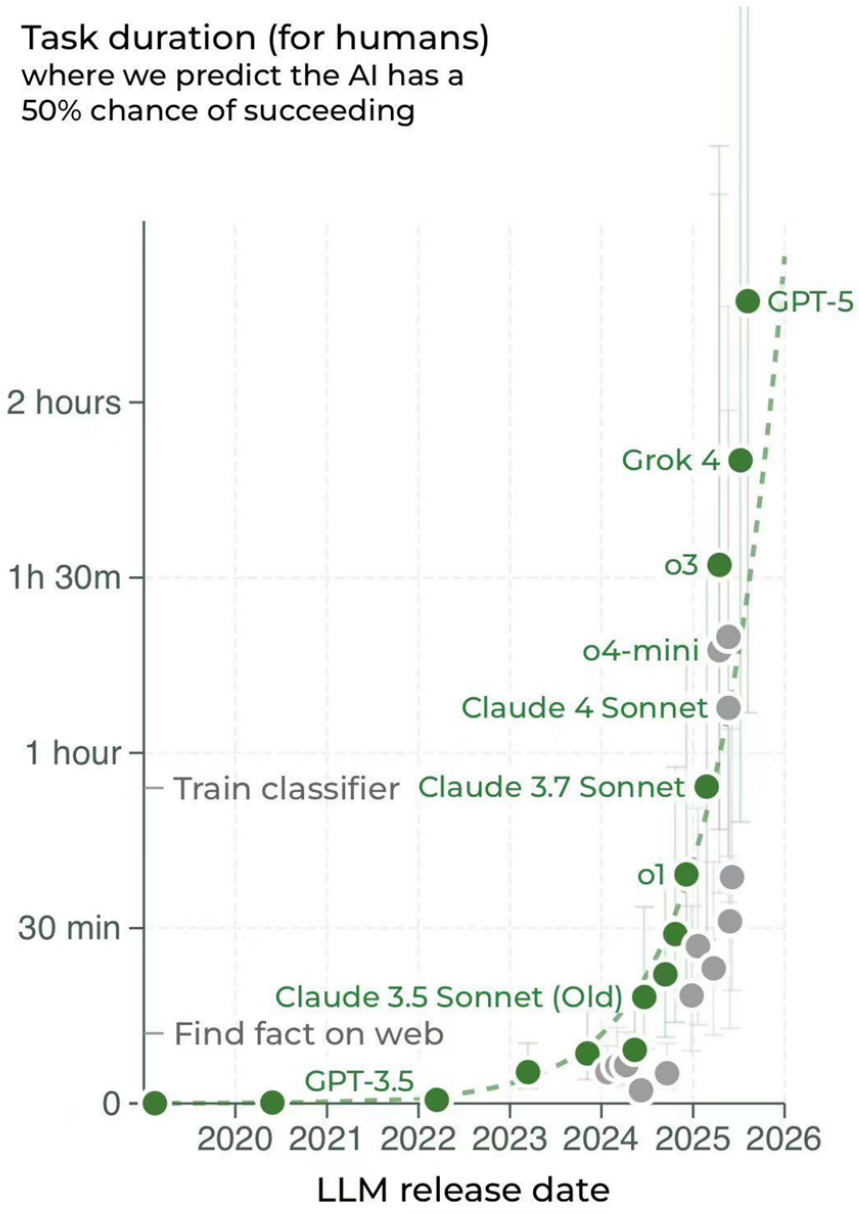

Figure 5.3: A graph showing the exponential increase in task length achievable by AI agents over time.

Bibliography

[20] *Neural Networks and Deep Learning*. http://neuralnetworksanddeeplearning.com/chap1.html. Accessed: 2025-07-26. 2020.

[LSC05] Quoc Le, Alexander Smola, and Stéphane Canu. "Heteroscedastic Gaussian process regression". In: Jan. 2005, pp. 489–496. DOI: 10.1145/1102351.1102413.

[Man20] Dominic Mangh. *Modeling Threshold Logic Neurons and Perceptrons*. Accessed: 2025-08-10. Aug. 2020. URL: https://dominicm73.blogspot.com/2020/08/modeling-threshold-logic-neurons-and.html.

[MP43] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.